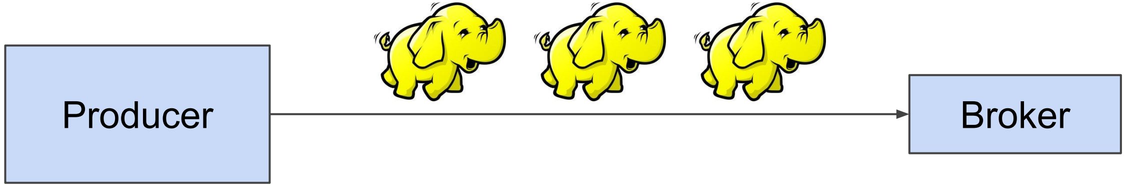
Handle Large Messages In Apache Kafka

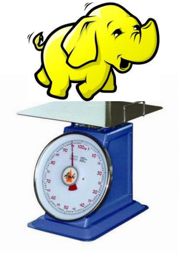
Jiangjie (Becket) Qin @ LinkedIn

Kafka Meetup - Feb 23, 2016

What is a “large message” ?

- Kafka has a limit on the maximum size of a single message
    - Enforced on the compressed wrapper message if compression is used





```
{
   …
   if (message.size > message.max.bytes)
     reject!
   …
}
```

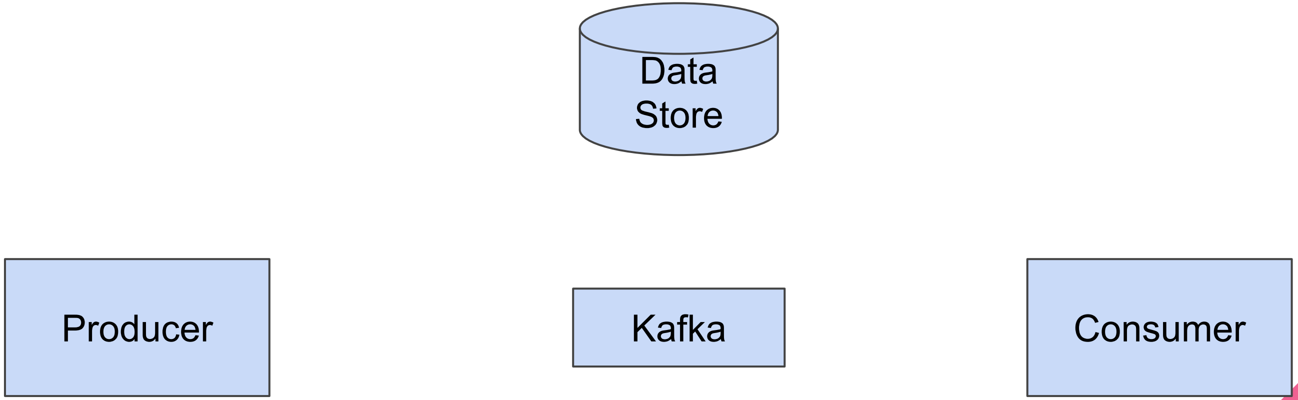
RecordTooLargeException

Why does Kafka limit the message size?

- Increase the **memory pressure** in the broker
- Large messages are **expensive** to handle and could **slow down the brokers**.
- A reasonable message size limit can handle vast **majority of the use cases**.

Why does Kafka limit the message size?

- Increase the **memory pressure** in the broker
- Large messages are **expensive** to handle and could **slow down the brokers**.
- A reasonable message size limit can handle vast **majority of the use cases**.
- Good workarounds exist (**Reference Based Messaging**)



Data Store

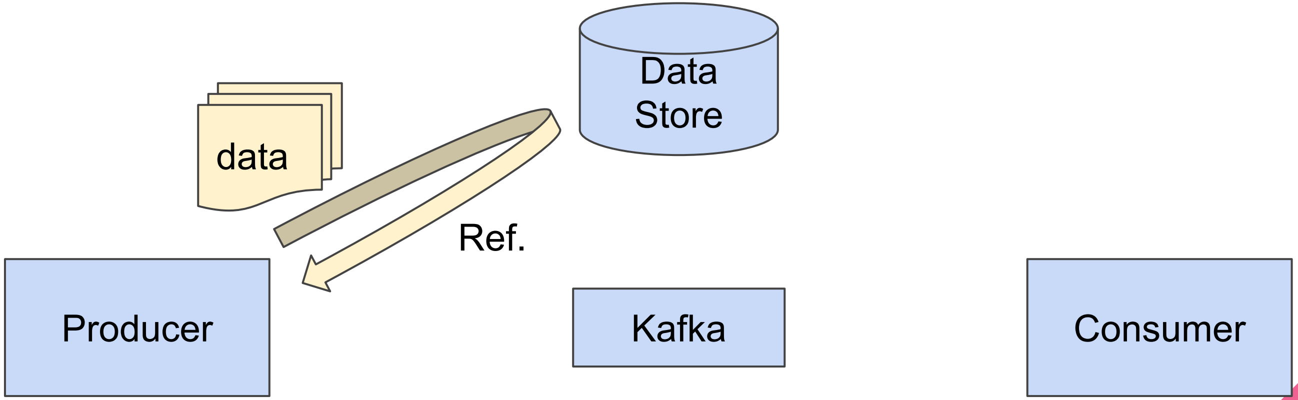
Producer

Kafka

Consumer

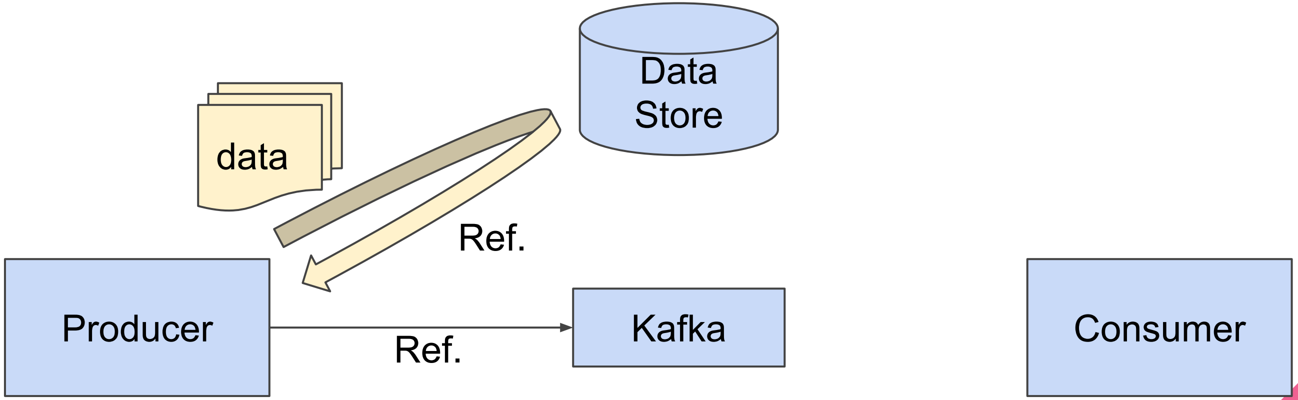
Why does Kafka limit the message size?

- Increase the **memory pressure** in the broker
- Large messages are **expensive** to handle and could **slow down the brokers**.
- A reasonable message size limit can handle vast **majority of the use cases**.
- Good workarounds exist (**Reference Based Messaging**)



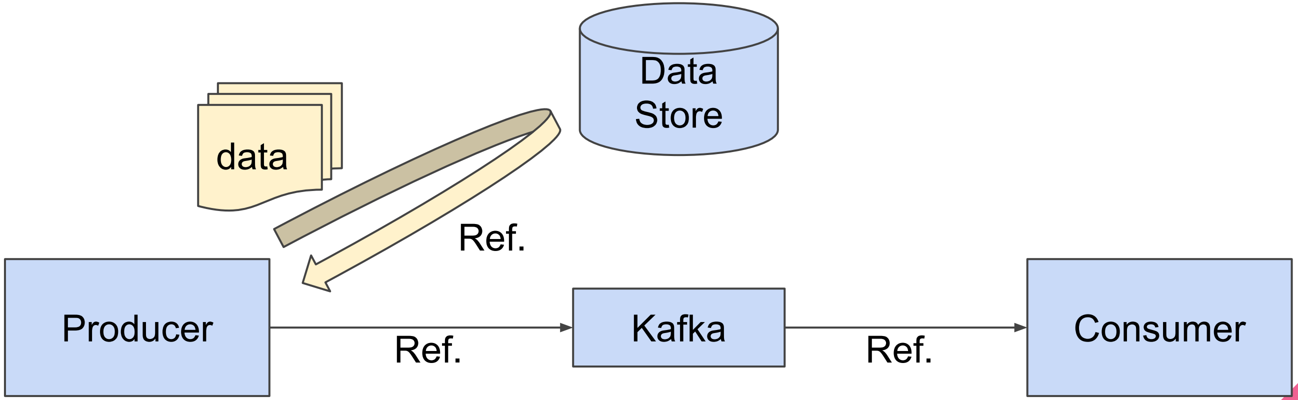
Why does Kafka limit the message size?

- Increase the **memory pressure** in the broker
- Large messages are **expensive** to handle and could **slow down the brokers**.
- A reasonable message size limit can handle vast **majority of the use cases**.
- Good workarounds exist (**Reference Based Messaging**)



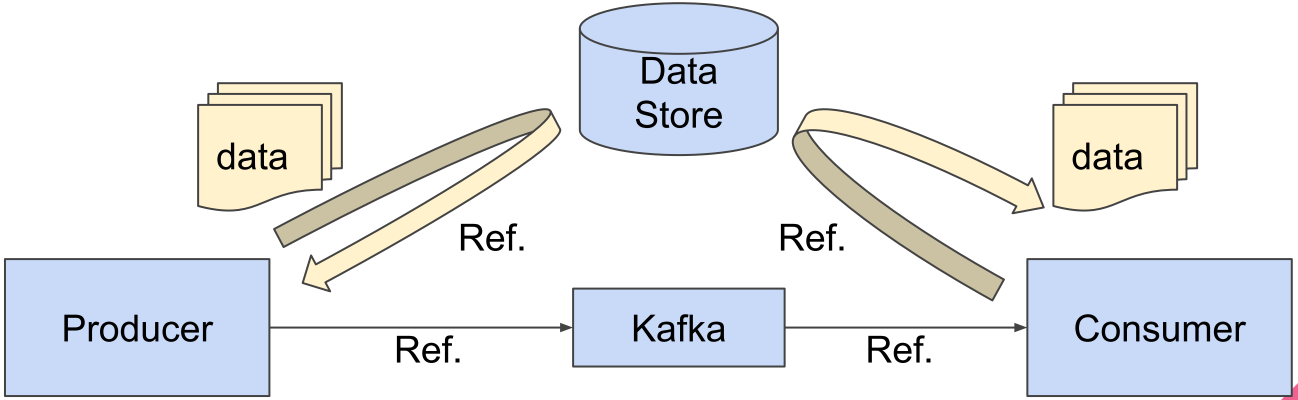
Why does Kafka limit the message size?

- Increase the **memory pressure** in the broker
- Large messages are **expensive** to handle and could **slow down the brokers**.
- A reasonable message size limit can handle vast **majority of the use cases**.
- Good workarounds exist (**Reference Based Messaging**)



Why does Kafka limit the message size?

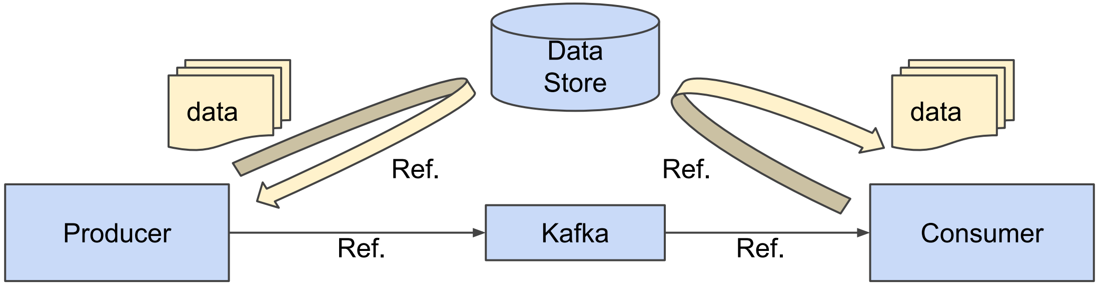
- Increase the **memory pressure** in the broker
- Large messages are **expensive** to handle and could **slow down the brokers**.
- A reasonable message size limit can handle vast **majority of the use cases**.
- Good workarounds exist (**Reference Based Messaging**)



Reference Based Messaging

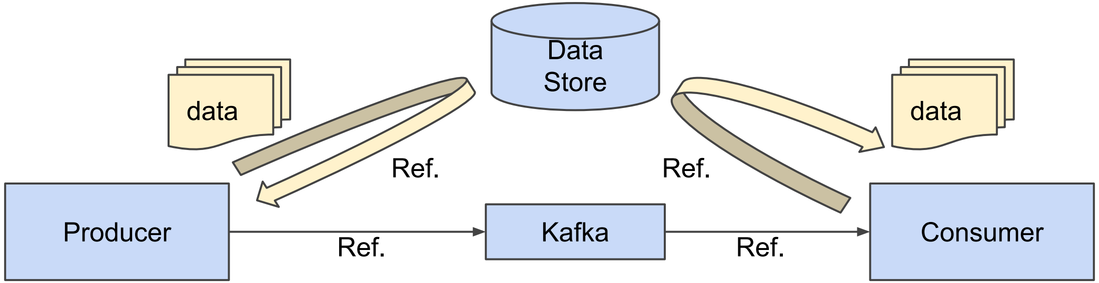
- One of our use cases: database replication
  - Unknown maximum row size
  - Strict no data loss
  - Strict message order guarantee

Works fine as long as the durability of the data store can be guaranteed.



Reference Based Messaging

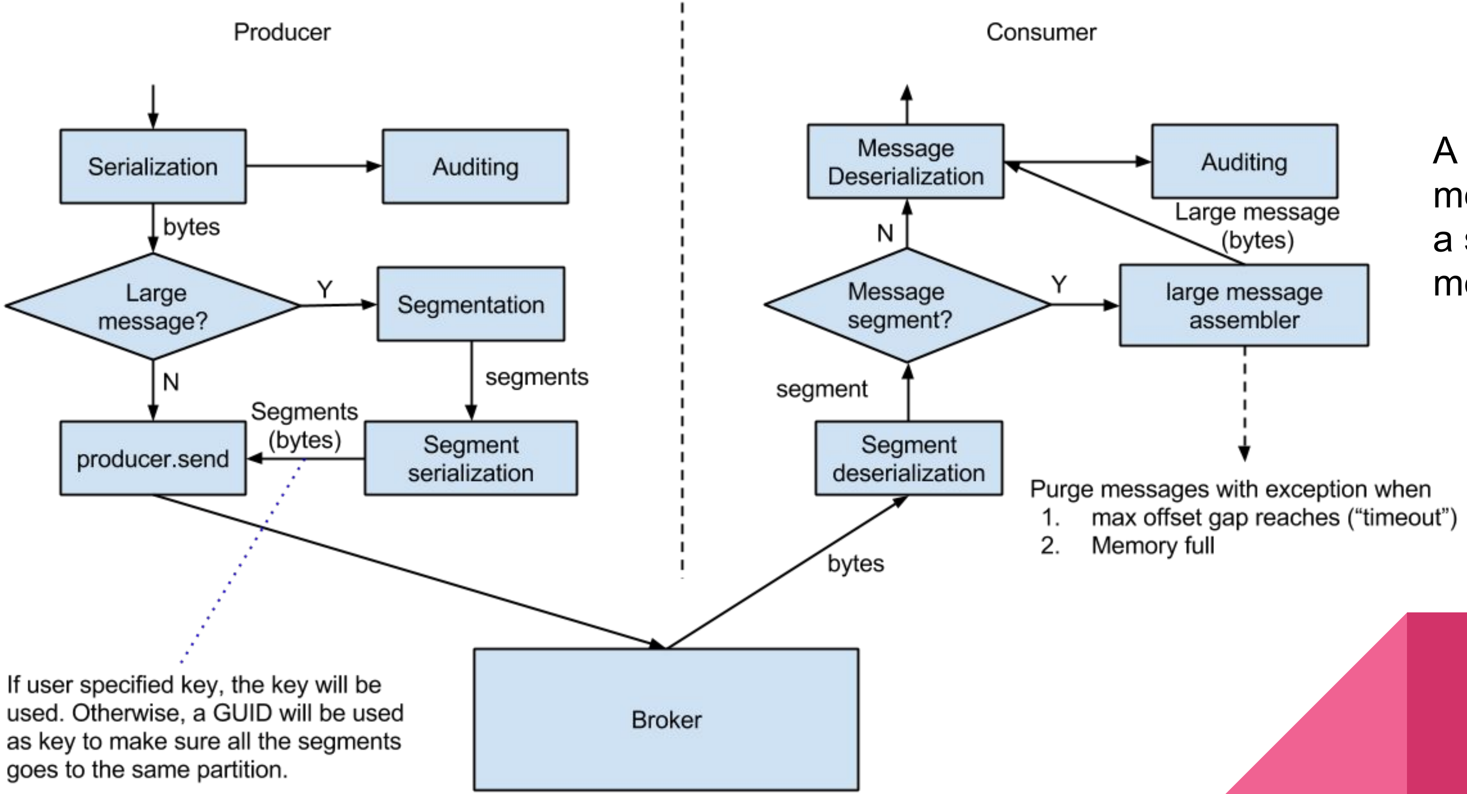
- One of our use cases: database replication
  - Replicates a data store by using another data store....
  - Sporadic large messages
    - Option 1: Send all the messages using reference and take unnecessary overhead.
    - Option 2: Only send large messages using references and live with low storage utilization.
  - Low end to end latency
    - There are more round trips in the system.
    - Need to make sure the data store is fast



In-line Large Message Support

	Reference Based Messaging	In-line large message support
Operational complexity	Two systems to maintain	Only maintain Kafka
System stability	Depend on : ● The consistency between Kafka and the external storage ● The durability of external storage	Only depend on Kafka
Cost to serve	Kafka + External Storage	Only maintain Kafka
End to end latency	Depend on the external storage	The latency of Kafka
Client complexity	Need to deal with envelopes	Much more involved (coming soon)
Functional limitations	Almost none	Some limitations

Our solution - chunk and re-assemble



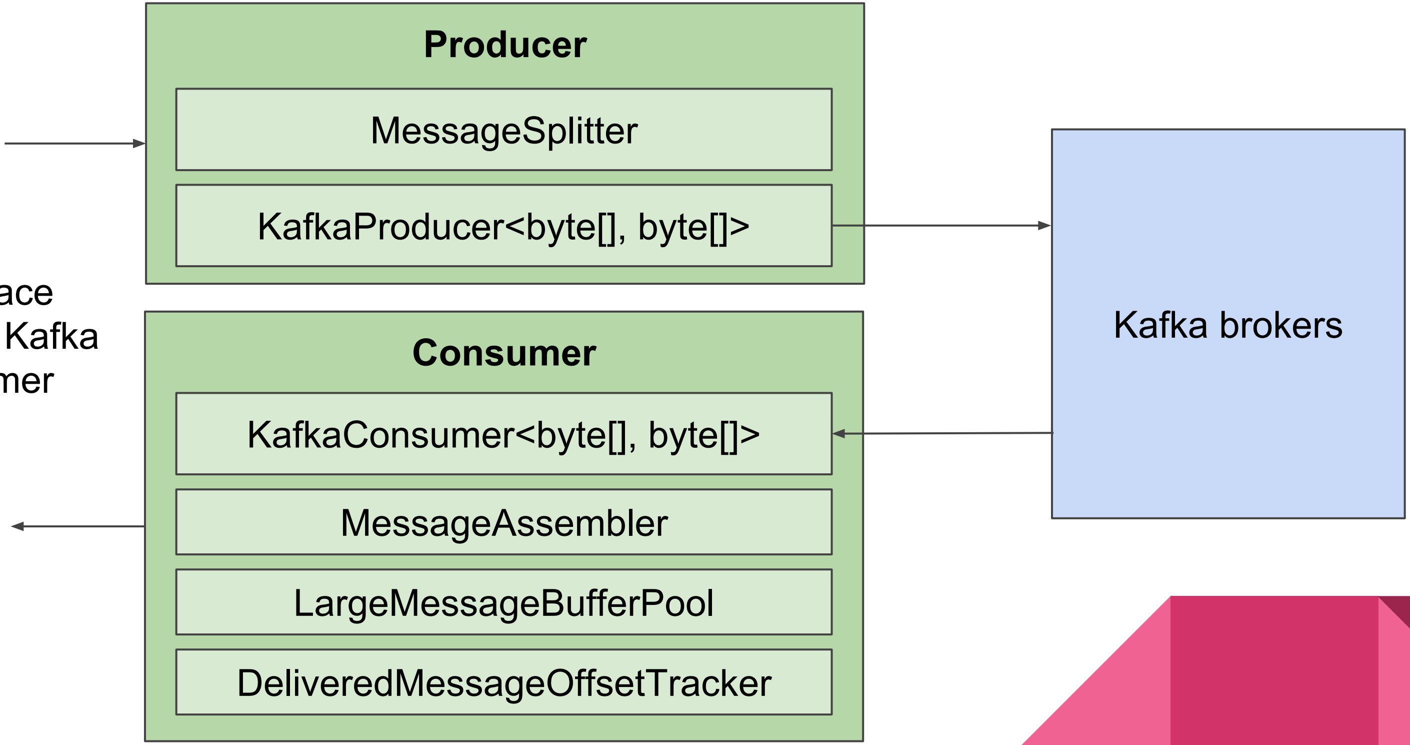
Purge messages with exception when

1. max offset gap reaches (“timeout”)
2. Memory full

If user specified key, the key will be used. Otherwise, a GUID will be used as key to make sure all the segments goes to the same partition.

A normal-sized message is sent as a single-segment message.

Client Modules


Producer
MessageSplitter
KafkaProducer<byte[], byte[]>
Consumer
KafkaConsumer<byte[], byte[]>
MessageAssembler
LargeMessageBufferPool
DeliveredMessageOffsetTracker
Kafka brokers

Compatible interface with open source Kafka producer / consumer

A closer look at large message handling

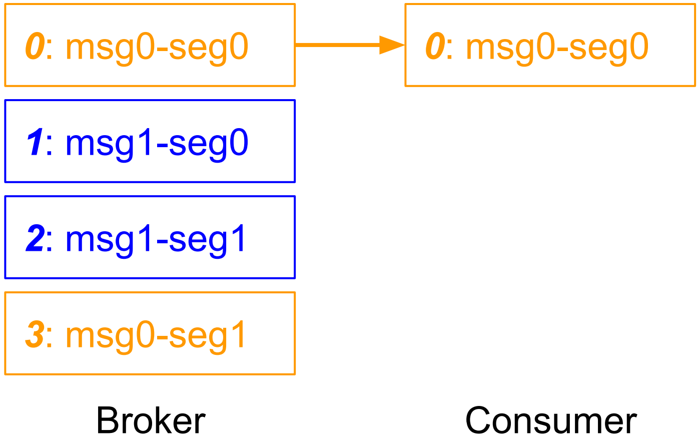
- The offset of a large message
- Producer callback
- Offset tracking
- Rebalance and duplicates handling
- Memory management
- Performance overhead
- Compatibility with existing messages

A closer look at large message handling

- **The offset of a large message**
- Offset tracking
- Producer callback
- Rebalance and duplicates handling
- Memory management
- Performance overhead
- Compatibility with existing messages

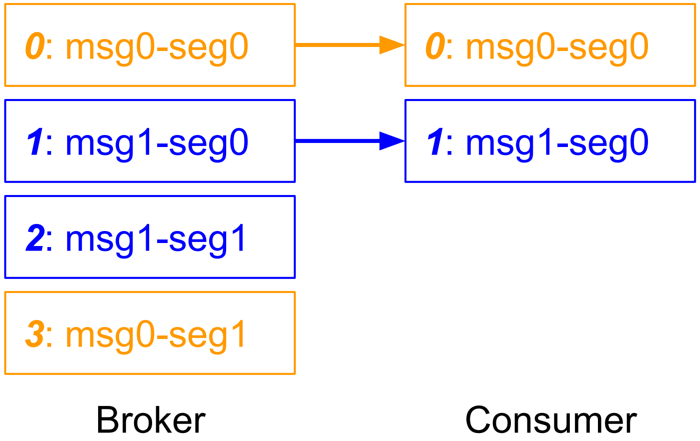
The offset of a large message

- The offset of the first segment?
  - **First seen first serve**
  - Easy to seek
  - Expensive for in order delivery



The offset of a large message

- The offset of the first segment?
  - **First seen first serve**
  - Easy to seek
  - Expensive for in order delivery



0: msg0-seg0

1: msg1-seg0

2: msg1-seg1

3: msg0-seg1

Broker

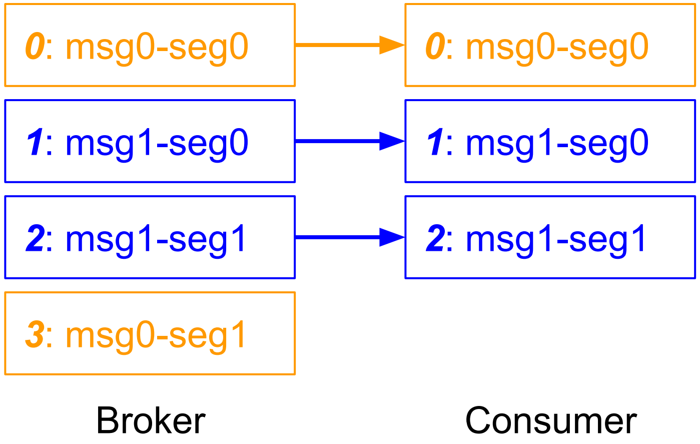
0: msg0-seg0

1: msg1-seg0

Consumer

The offset of a large message

- The offset of the first segment?
    - **First seen first serve**
    - Easy to seek
    - **Expensive for in order delivery**

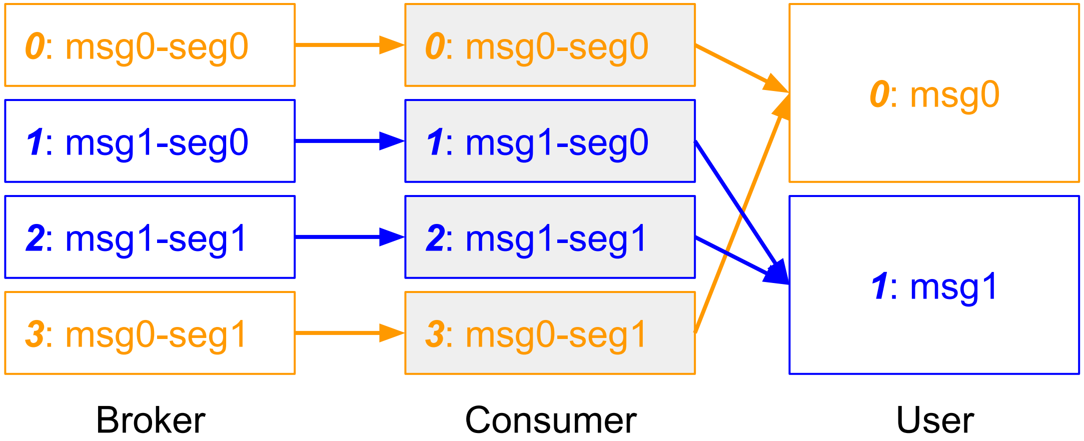


Cannot deliver msg1 until msg0 is delivered.
The consumer has to buffer the msg1.

Difficult to handle partially sent messages.

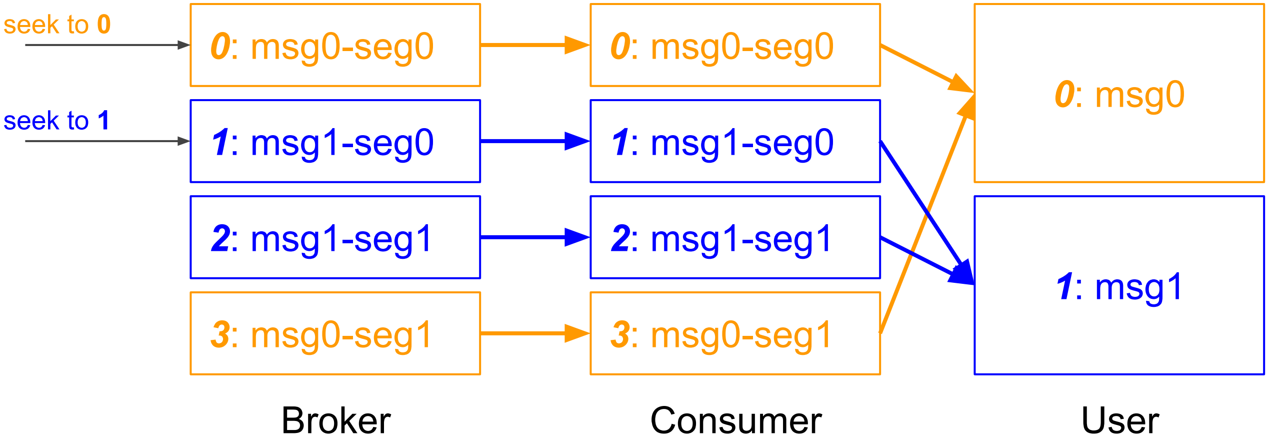
The offset of a large message

- The offset of the first segment?
  - **First seen first serve**
  - Easy to seek
  - Expensive for in order delivery (Need to buffer all the message segments until the current large message is complete)



The offset of a large message

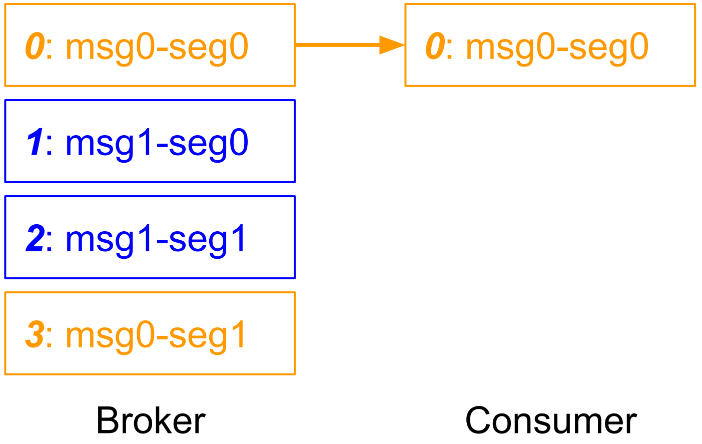
- The offset of the first segment?
  - First seen first serve
  - **Easy to seek**
  - Expensive for in order delivery



The consumer can simply seek to the message offset.

The offset of a large message

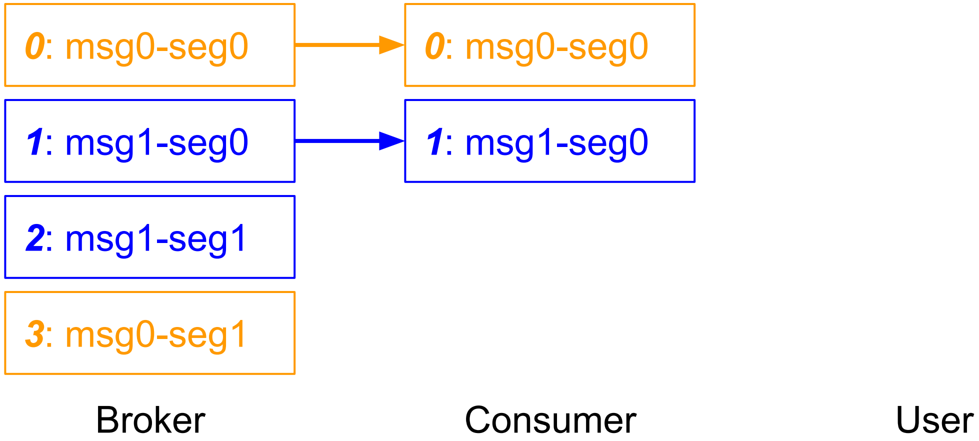
- The offset of the last segment?
  - **First completed first serve**
  - Needs additional work for seek (more details on this soon)
  - Least memory needed for in order delivery
- We chose offset of the last segment



User

The offset of a large message

- The offset of the last segment?
  - **First completed first serve**
  - Needs additional work for seek (more details on this soon)
  - Least memory needed for in order delivery
- We chose offset of the last segment



0: msg0-seg0

1: msg1-seg0

2: msg1-seg1

3: msg0-seg1

0: msg0-seg0

1: msg1-seg0

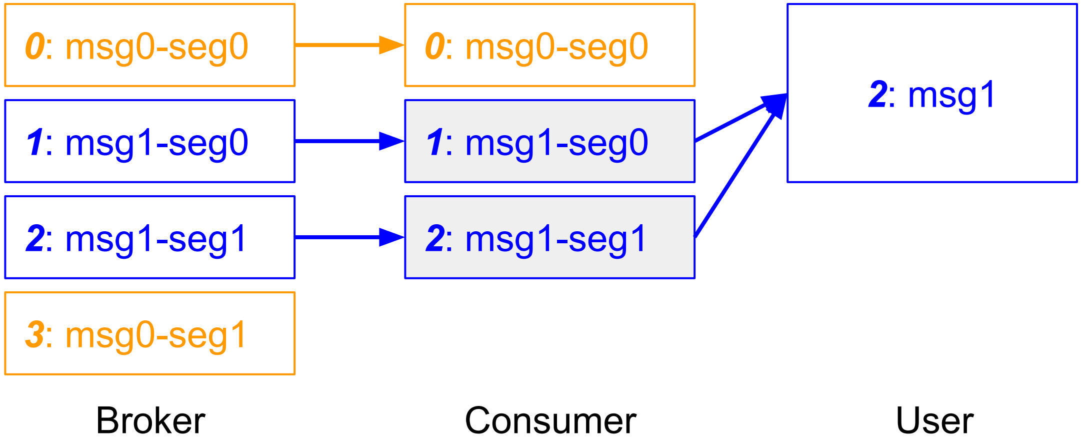
Broker

Consumer

User

The offset of a large message

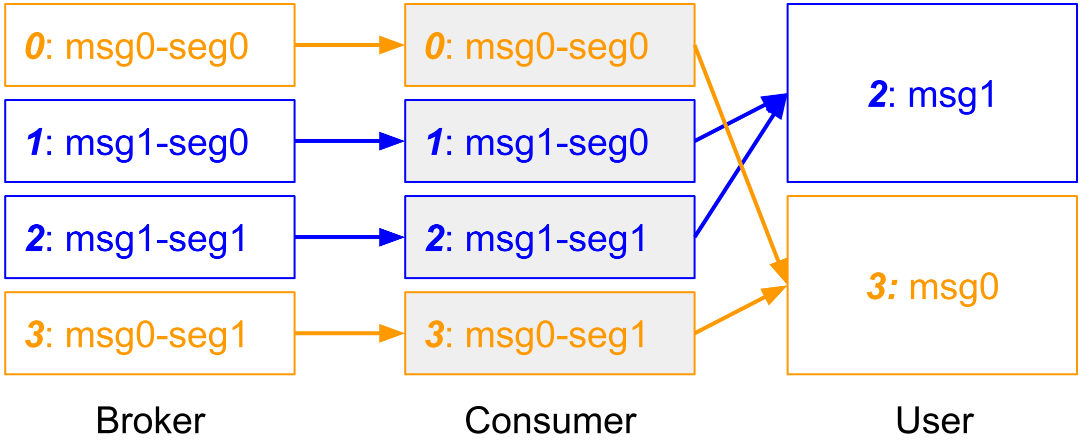
- The offset of the last segment?
  - **First completed first serve**
  - Needs additional work for seek (more details on this soon)
  - **Least memory needed for in order delivery**
- We chose offset of the last segment



Deliver msg1 once it completes.

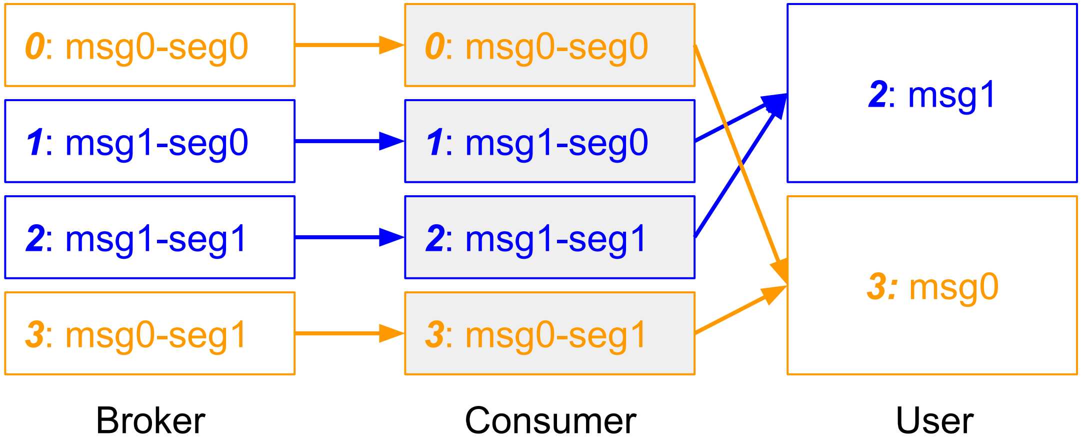
The offset of a large message

- The offset of the last segment?
  - First completed first serve
  - **Needs additional work for seek** (more details in offset tracking)
  - Least memory needed for in order delivery



The offset of a large message

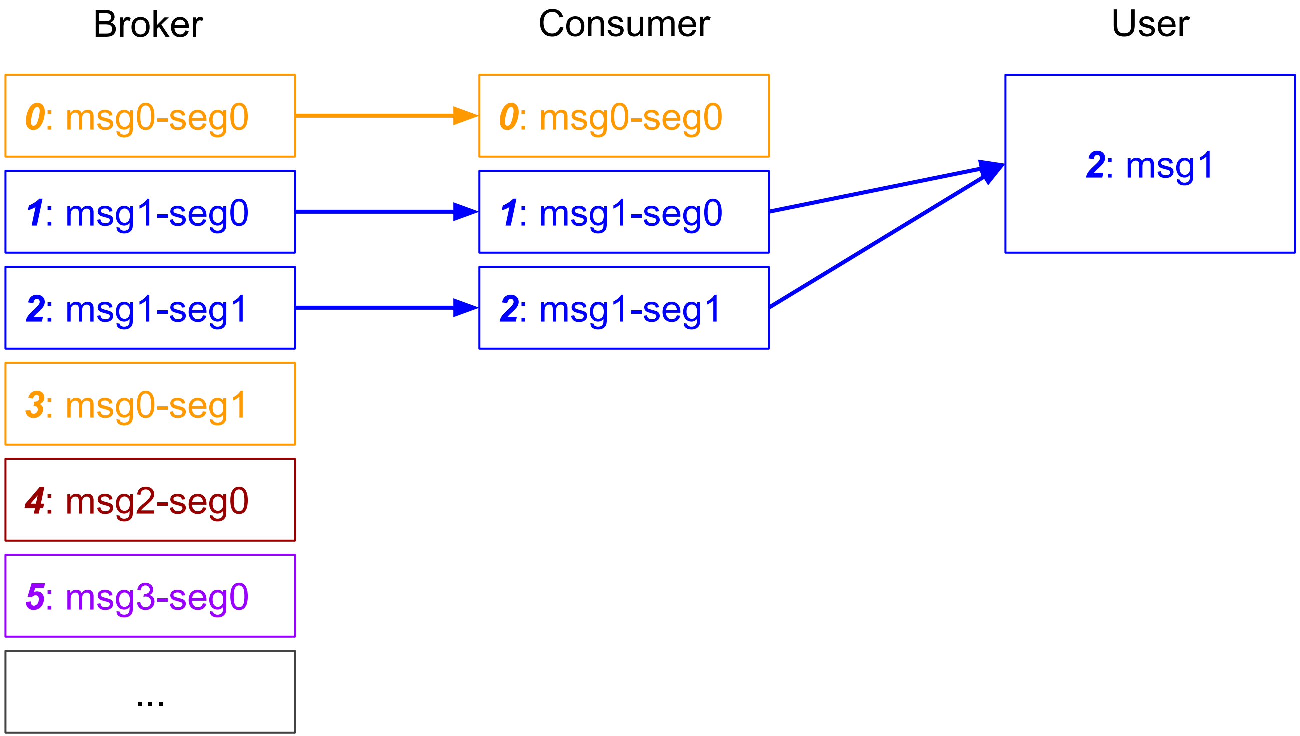
- **We chose offset of the last segment**
  - Less memory consumption
  - Better tolerance for partially sent large messages.



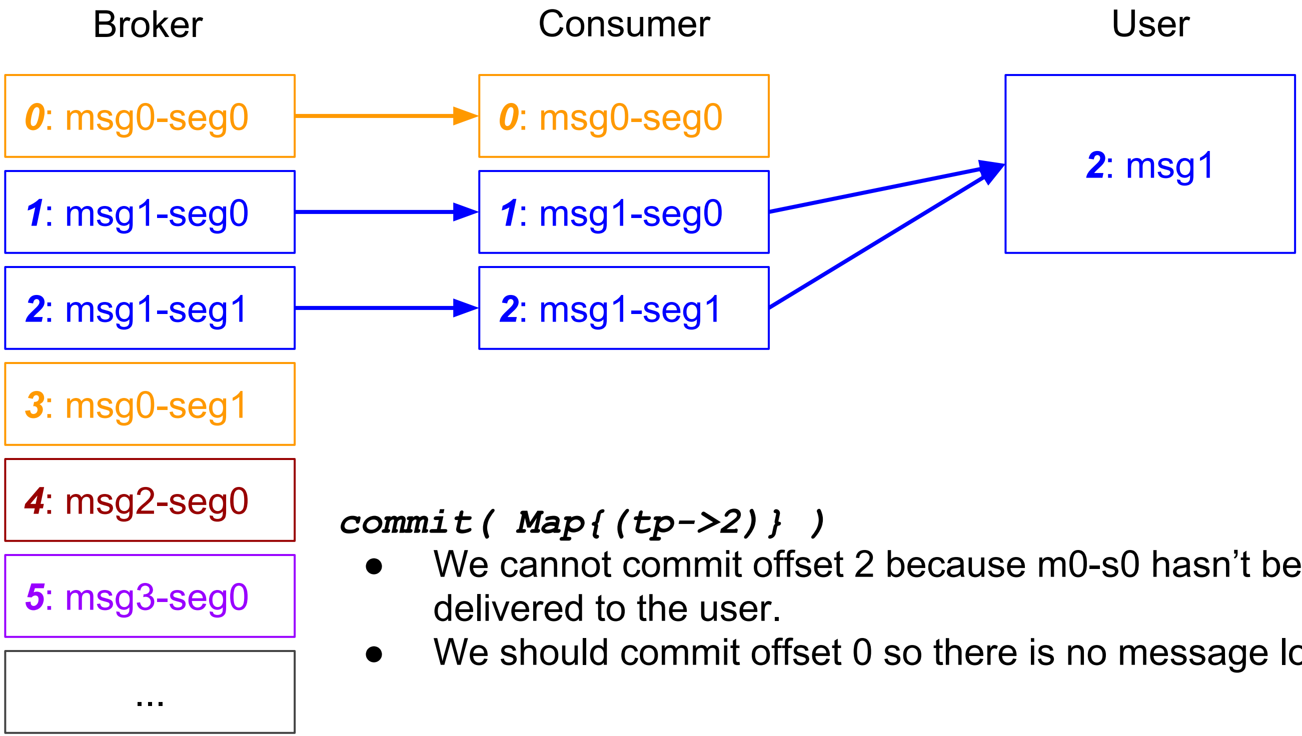
A closer look at large message handling

- The offset of a large message
- **Offset tracking**
- Producer callback
- Rebalance and duplicates handling
- Memory management
- Performance overhead
- Compatibility with existing messages

Offset tracking



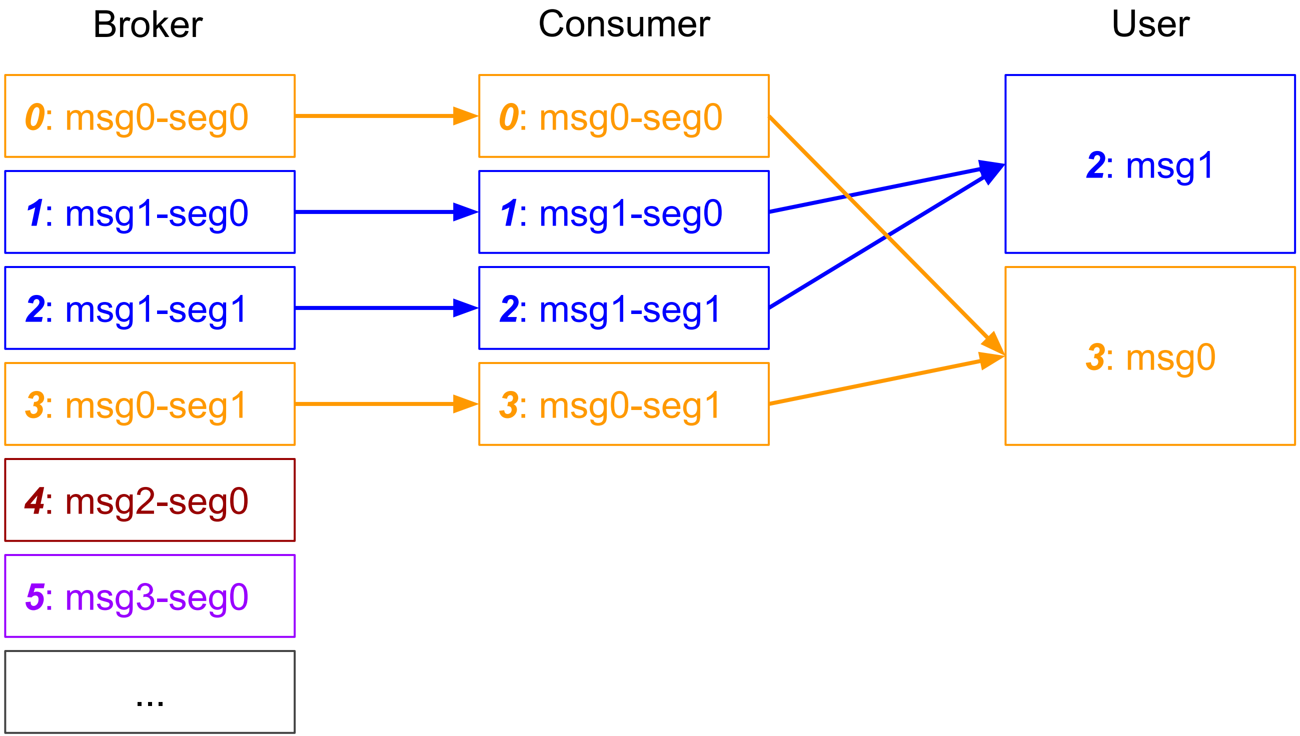
Offset tracking



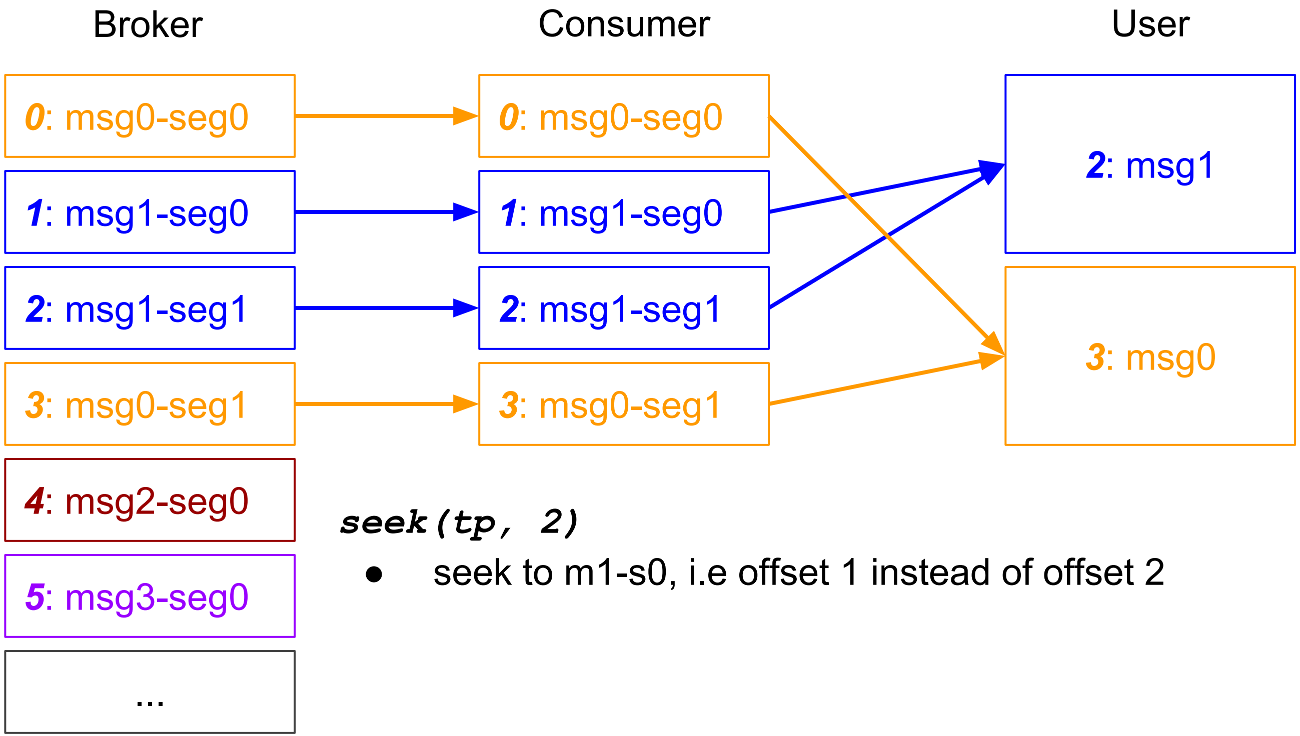
commit(Map{(tp->2)})

- We cannot commit offset 2 because m0-s0 hasn’t been delivered to the user.
- We should commit offset 0 so there is no message loss.

Offset tracking



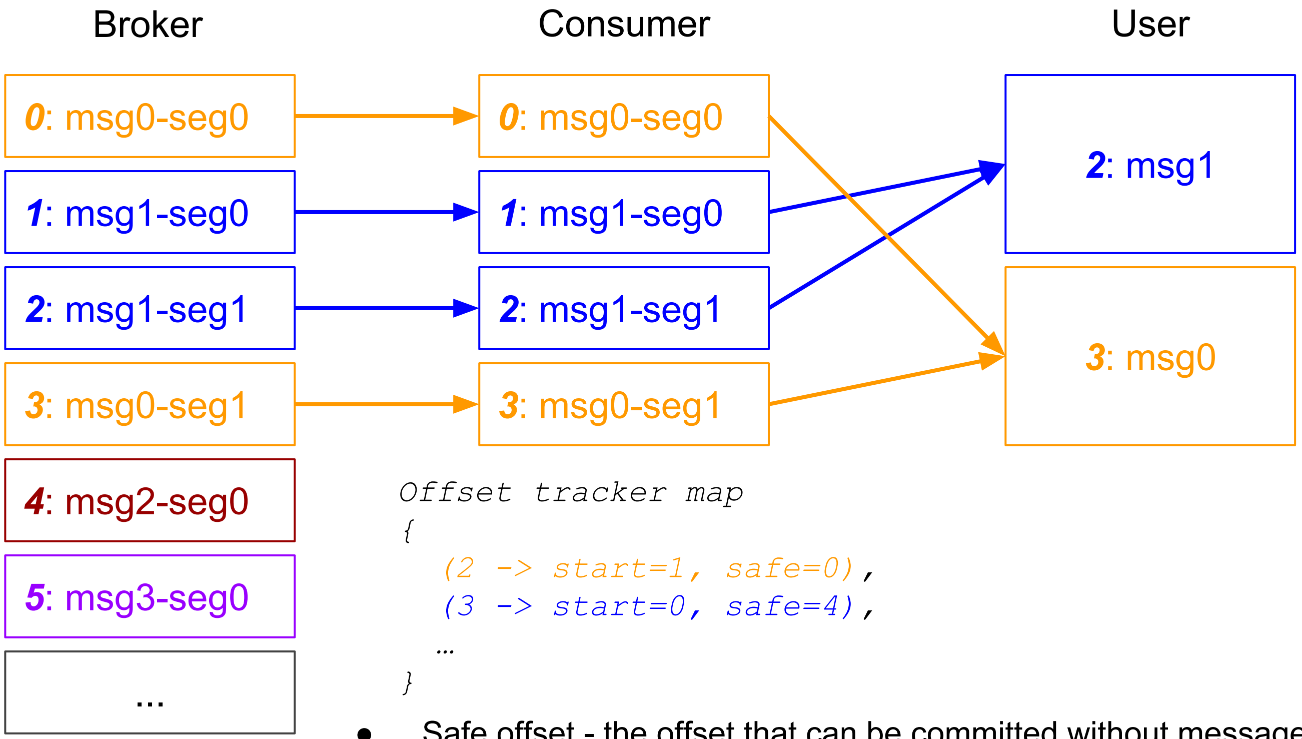
Offset tracking



`seek(tp, 2)`

- seek to m1-s0, i.e offset 1 instead of offset 2

Offset tracking



```
Offset tracker map
{
   (2 -> start=1, safe=0),
   (3 -> start=0, safe=4),
   …
}
```

- Safe offset - the offset that can be committed without message loss
- Starting offset - the starting offset of a large message.

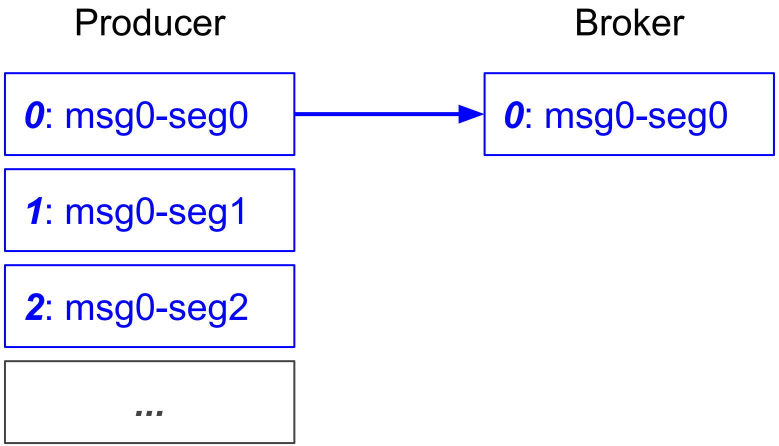
Offset tracking

- Limitations
    - Consumers can only track the message they have already seen.
        - When the users seek forward, consumer does not check if user is seeking to a message boundary.
    - Consumers cannot keep track of all the messages they have ever seen.
        - Consumers only track a configured number of recently delivered message for each partition. e.g. 5,000.
    - After rebalance, the new owner of a partition will not have any tracked message from the newly assigned partitions.

A closer look at large message handling

- The offset of a large message
- Offset tracking
- **Producer callback**
- Rebalance and duplicates handling
- Memory management
- Performance overhead
- Compatibility with existing messages

Producer Callback



Producer

Broker

0: msg0-seg0

1: msg0-seg1

2: msg0-seg2

...

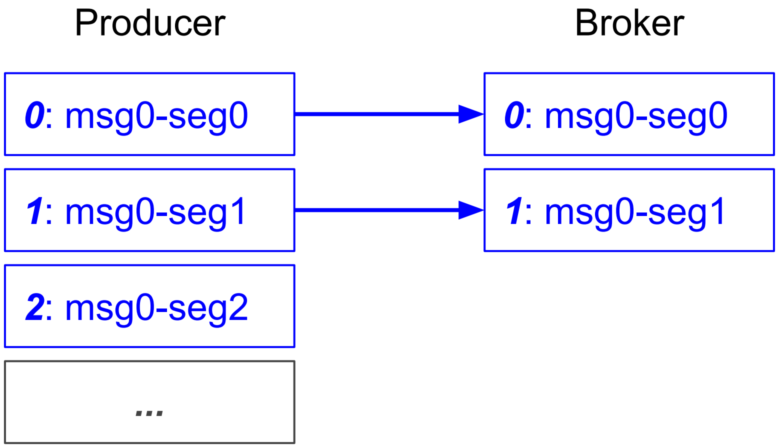
0: msg0-seg0

All the segments will be sent to the same partition.

```
{
    numSegments=3
    ackedSegments=1;
    userCallback;
}
```

Do not fire user callback

Producer Callback

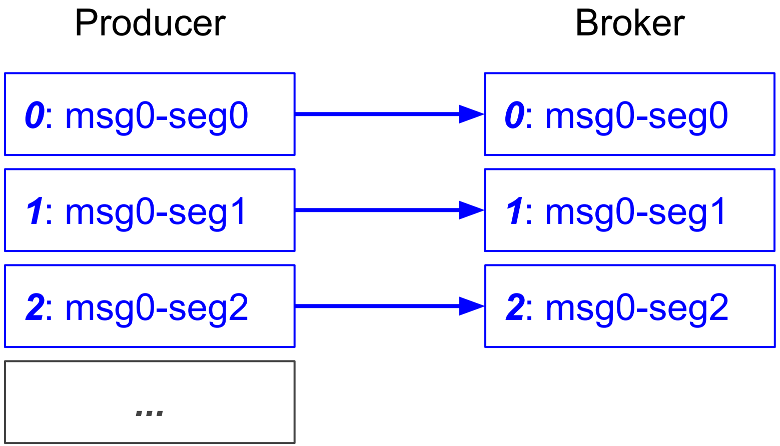


All the segments will be sent to the same partition.

```
{
    numSegments=3
    ackedSegments=2;
    userCallback;
}
```

Do not fire user callback

Producer Callback



Producer

Broker

0: msg0-seg0

1: msg0-seg1

2: msg0-seg2

...

0: msg0-seg0

1: msg0-seg1

2: msg0-seg2

All the segments will be sent to the same partition.

```
{
    numSegments=3
    ackedSegments=3;
    userCallback;
}
```

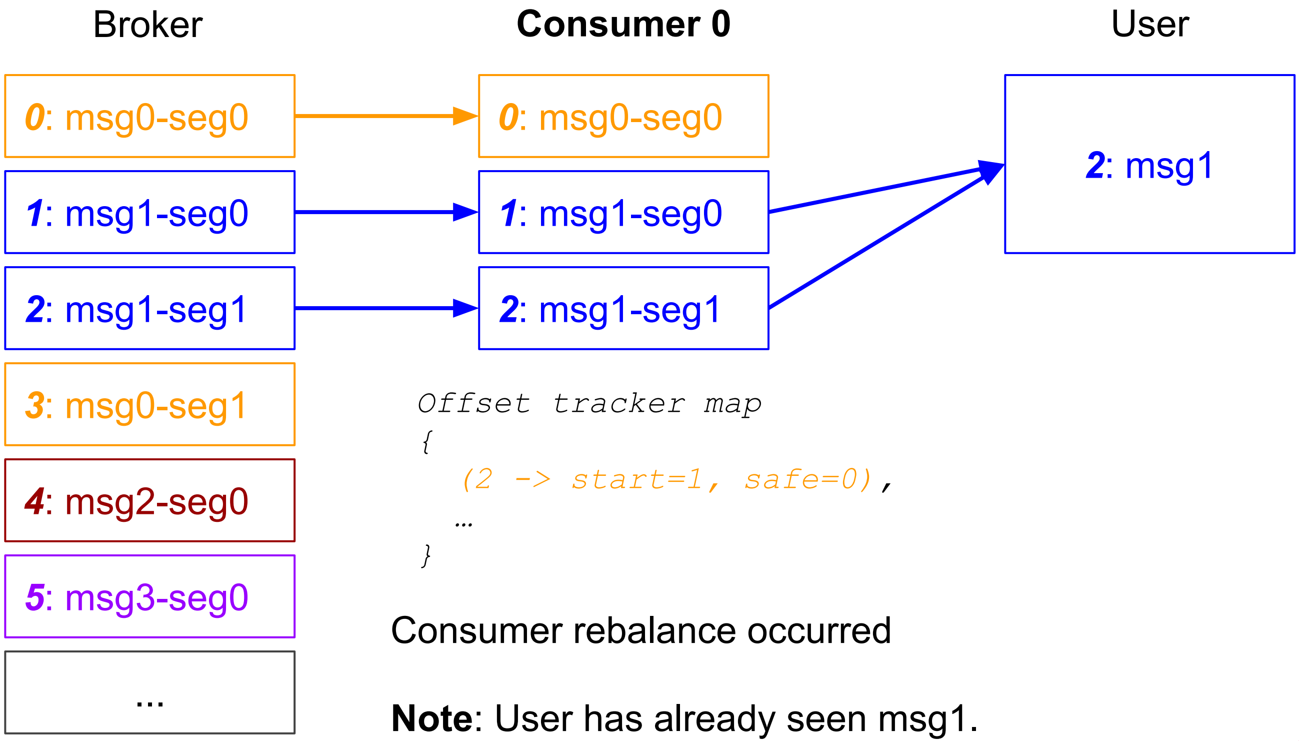
Fire the user callback

- The **offset of the last segment** is passed to the user callback
- The **first exception received** is passed to the user callback

A closer look at large message handling

- The offset of a large message
- Producer callback
- Offset tracking
- **Rebalance and duplicates handling**
- Memory management
- Performance overhead
- Compatibility with existing messages

Rebalance and duplicates handling

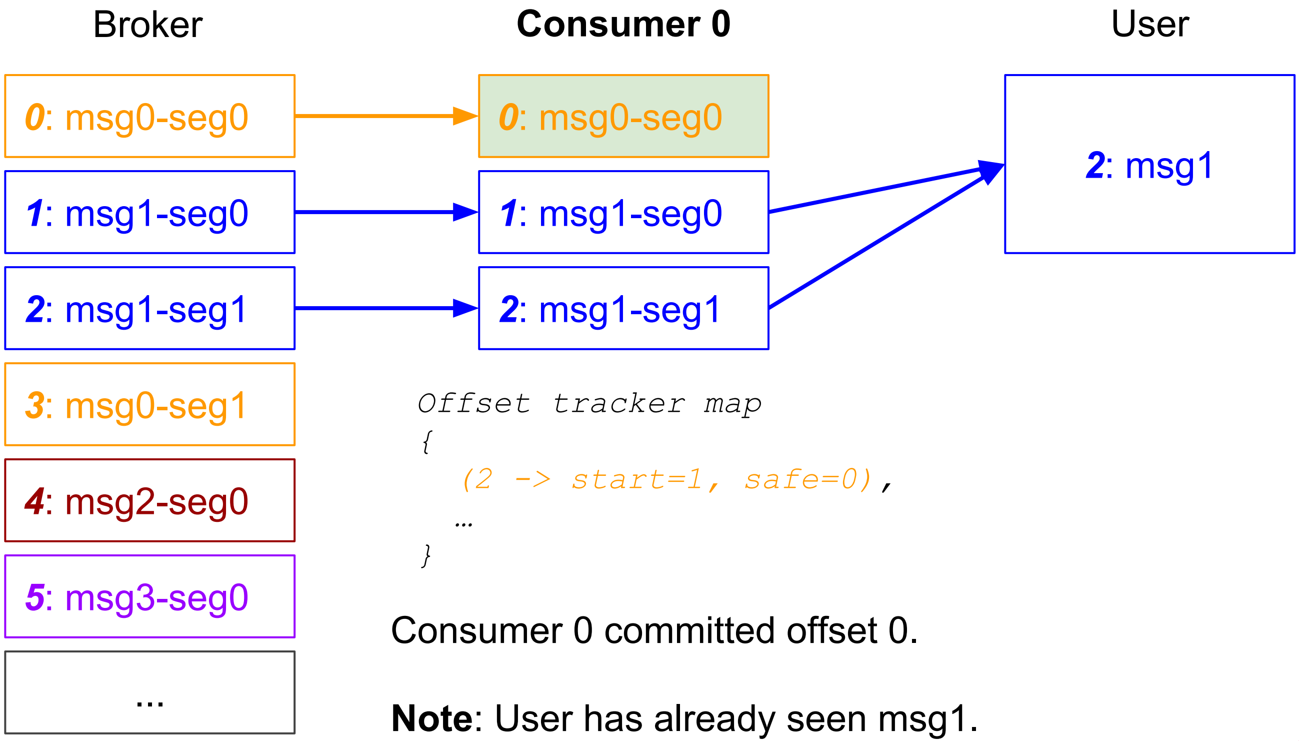


```
Offset tracker map
{
   (2 -> start=1, safe=0),
   …
}
```

Consumer rebalance occurred

Note: User has already seen msg1.

Rebalance and duplicates handling



Broker

Consumer 0

User

0: msg0-seg0

1: msg1-seg0

2: msg1-seg1

3: msg0-seg1

4: msg2-seg0

5: msg3-seg0

...

0: msg0-seg0

1: msg1-seg0

2: msg1-seg1

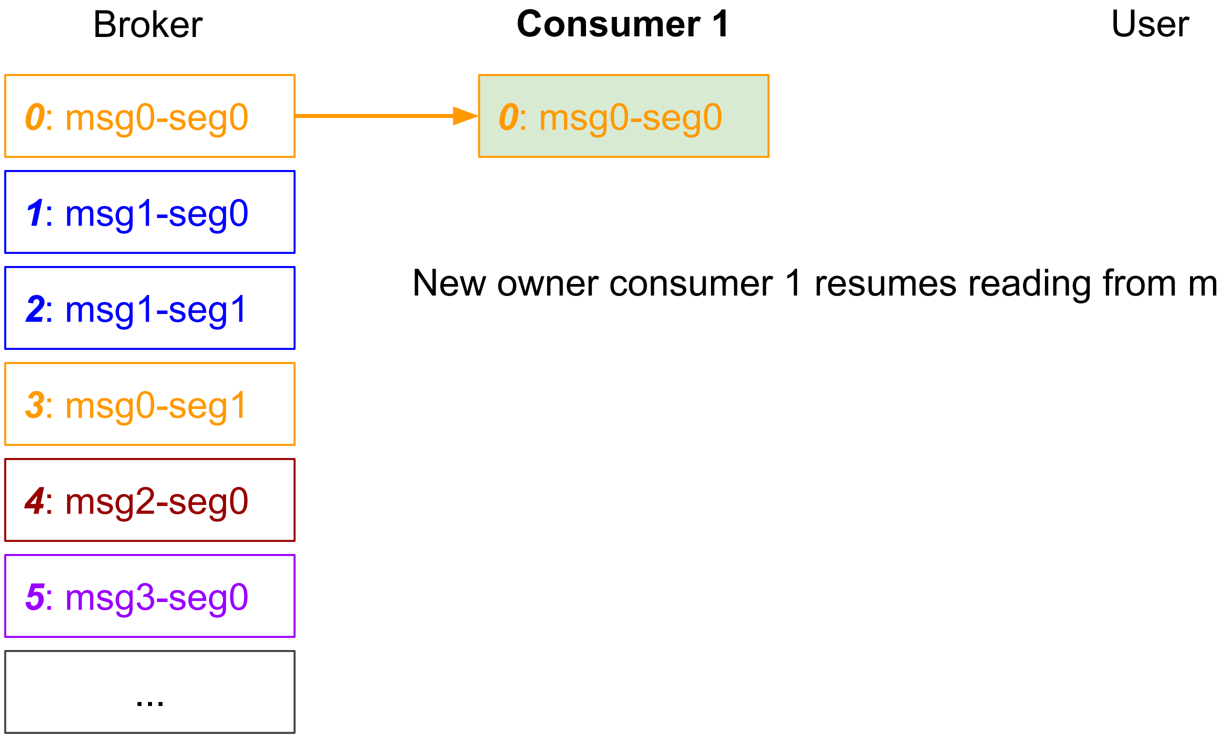
2: msg1

```
Offset tracker map
{
   (2 -> start=1, safe=0),
   …
}
```

Consumer 0 committed offset 0.

Note: User has already seen msg1.

Rebalance and duplicates handling



Broker

Consumer 1

User

0: msg0-seg0

1: msg1-seg0

2: msg1-seg1

3: msg0-seg1

4: msg2-seg0

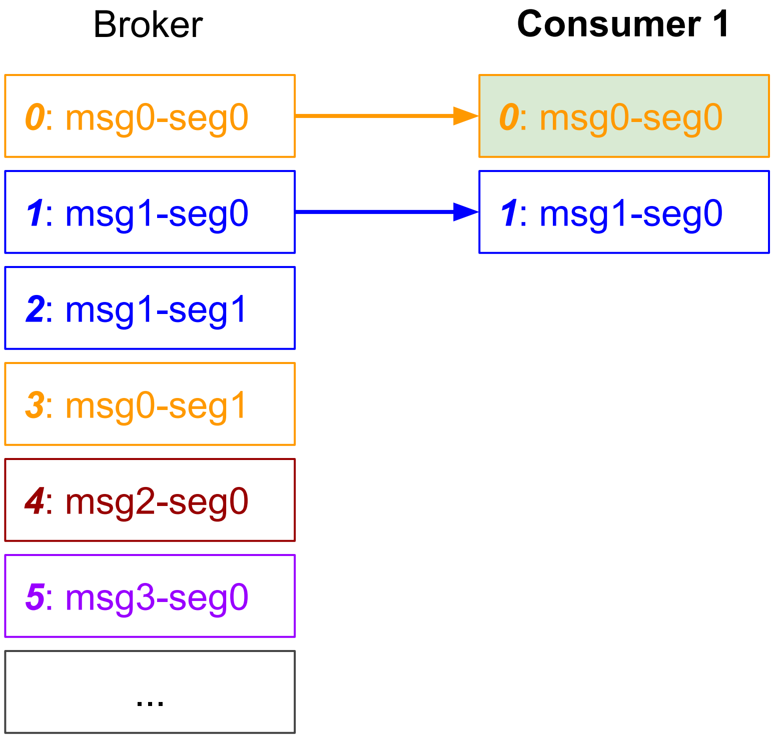
5: msg3-seg0

...

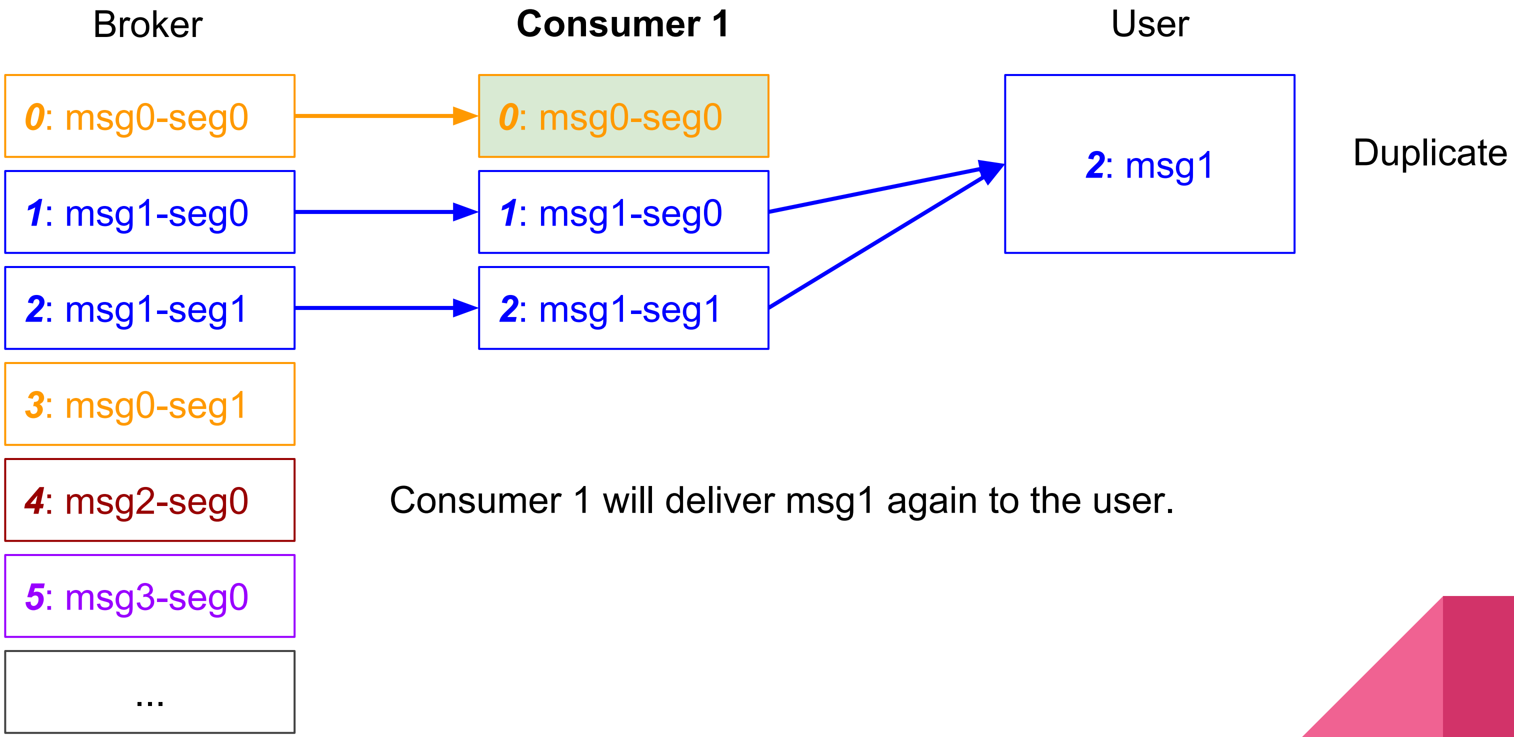
0: msg0-seg0

New owner consumer 1 resumes reading from msg0-seg0

Rebalance and duplicates handling

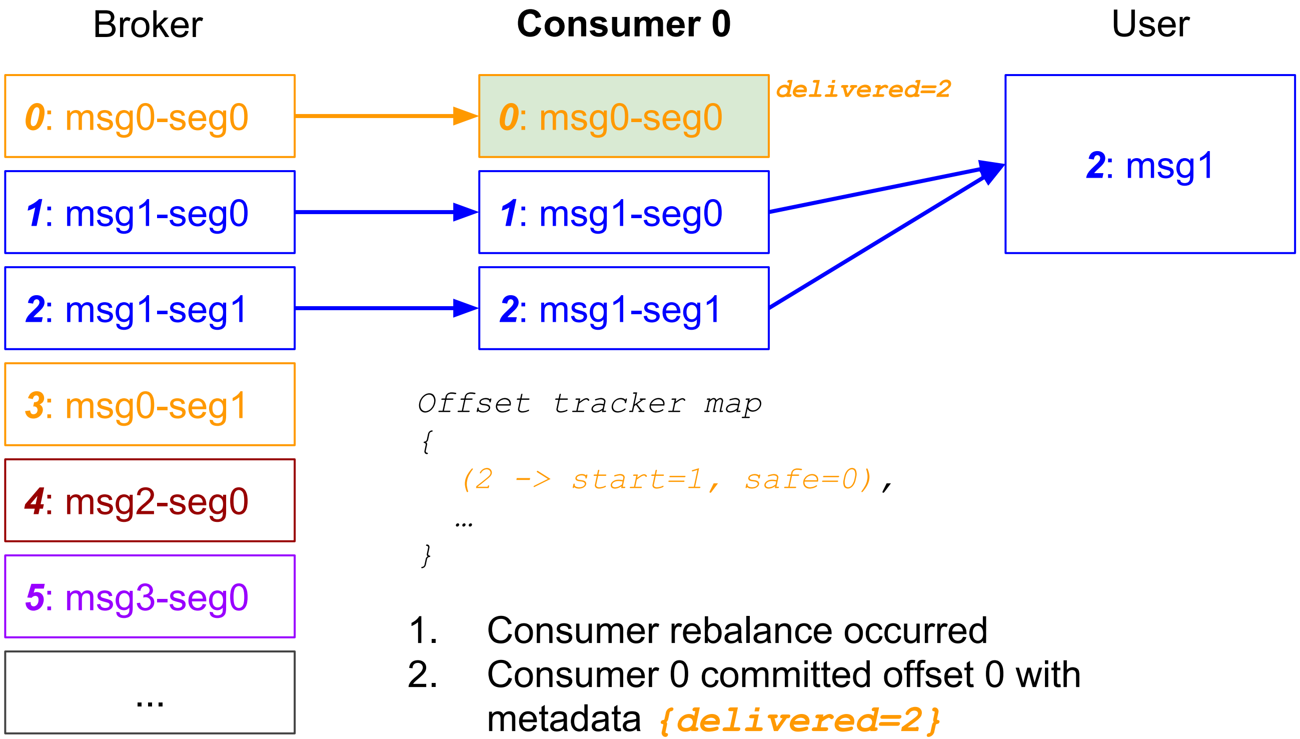


Rebalance and duplicates handling



Consumer 1 will deliver msg1 again to the user.

Rebalance and duplicates handling

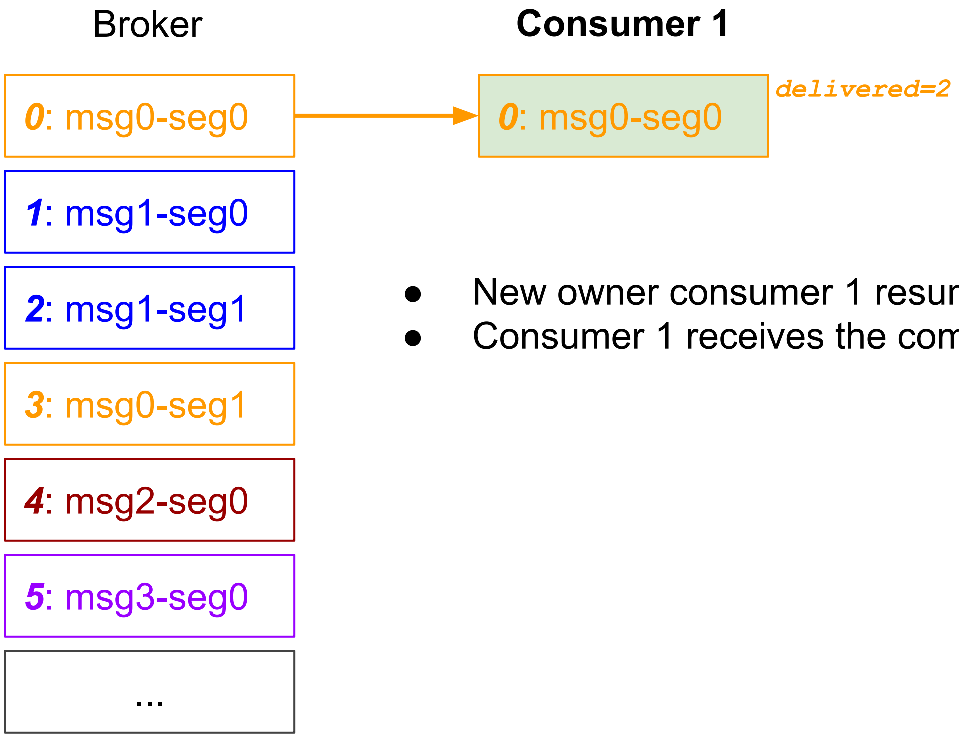


```
Offset tracker map
{
   (2 -> start=1, safe=0),
   …
}
```

1. Consumer rebalance occurred
2. Consumer 0 committed offset 0 with metadata *`{delivered=2}`*

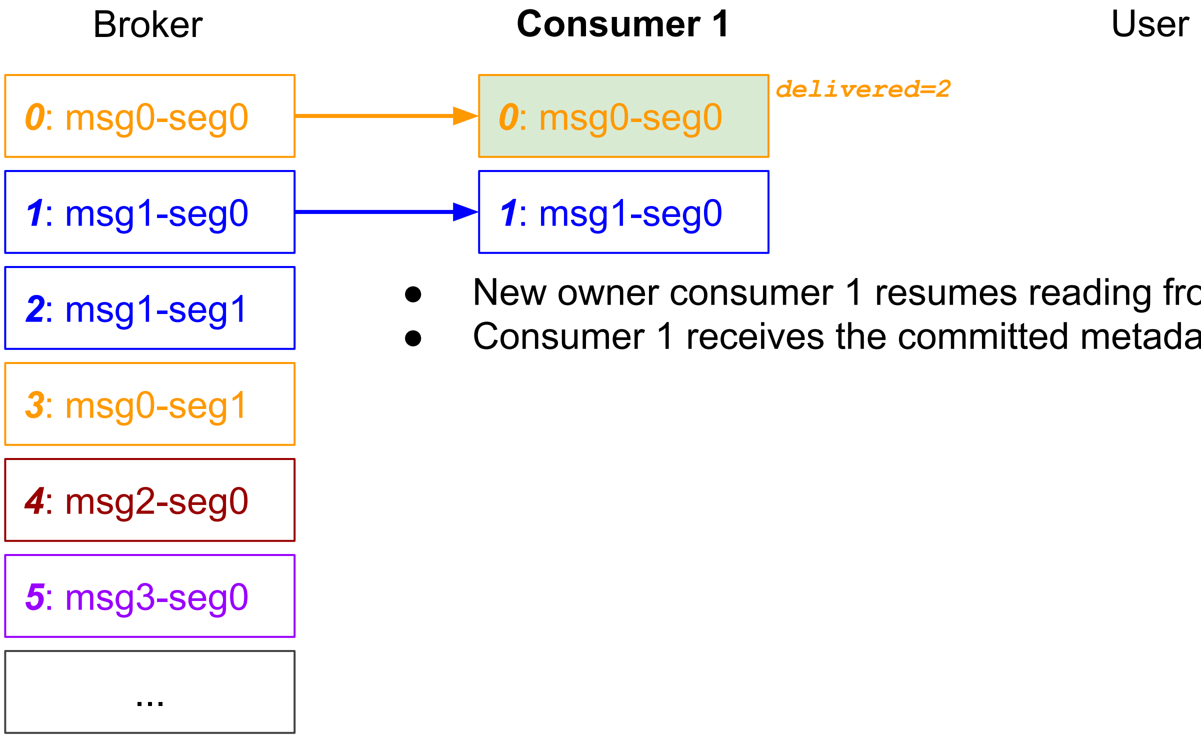
Note: User has already seen msg1.

Rebalance and duplicates handling



- New owner consumer 1 resumes reading from msg0-seg0
- Consumer 1 receives the committed metadata `{delivered=2}`

Rebalance and duplicates handling



Broker

Consumer 1

User

0: msg0-seg0

1: msg1-seg0

2: msg1-seg1

3: msg0-seg1

4: msg2-seg0

5: msg3-seg0

...

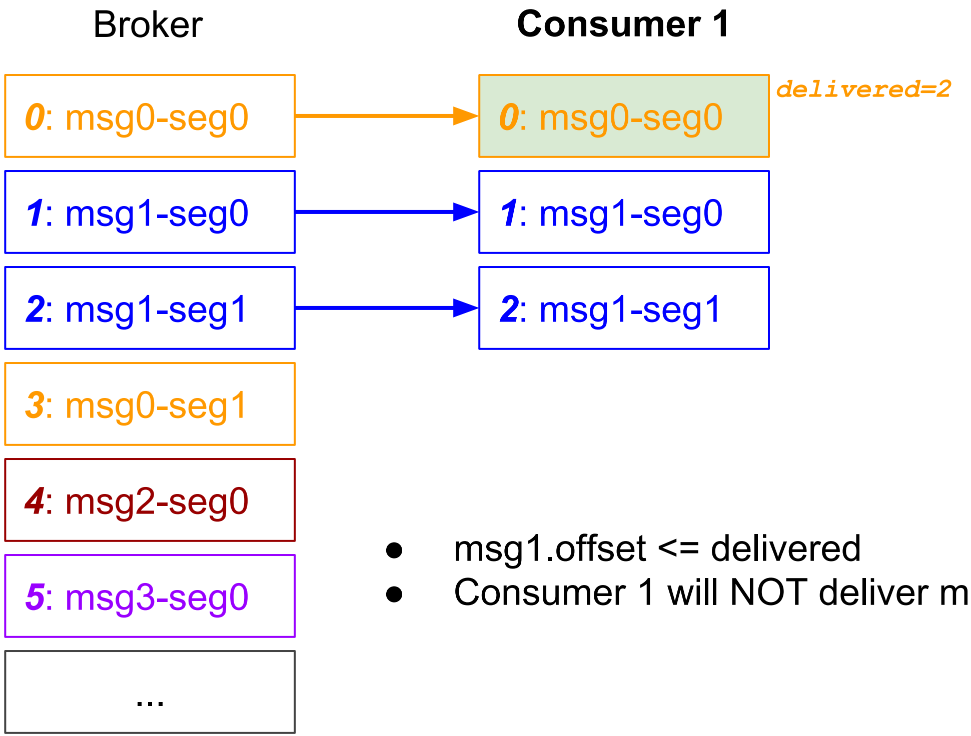
0: msg0-seg0

`delivered=2`

1: msg1-seg0

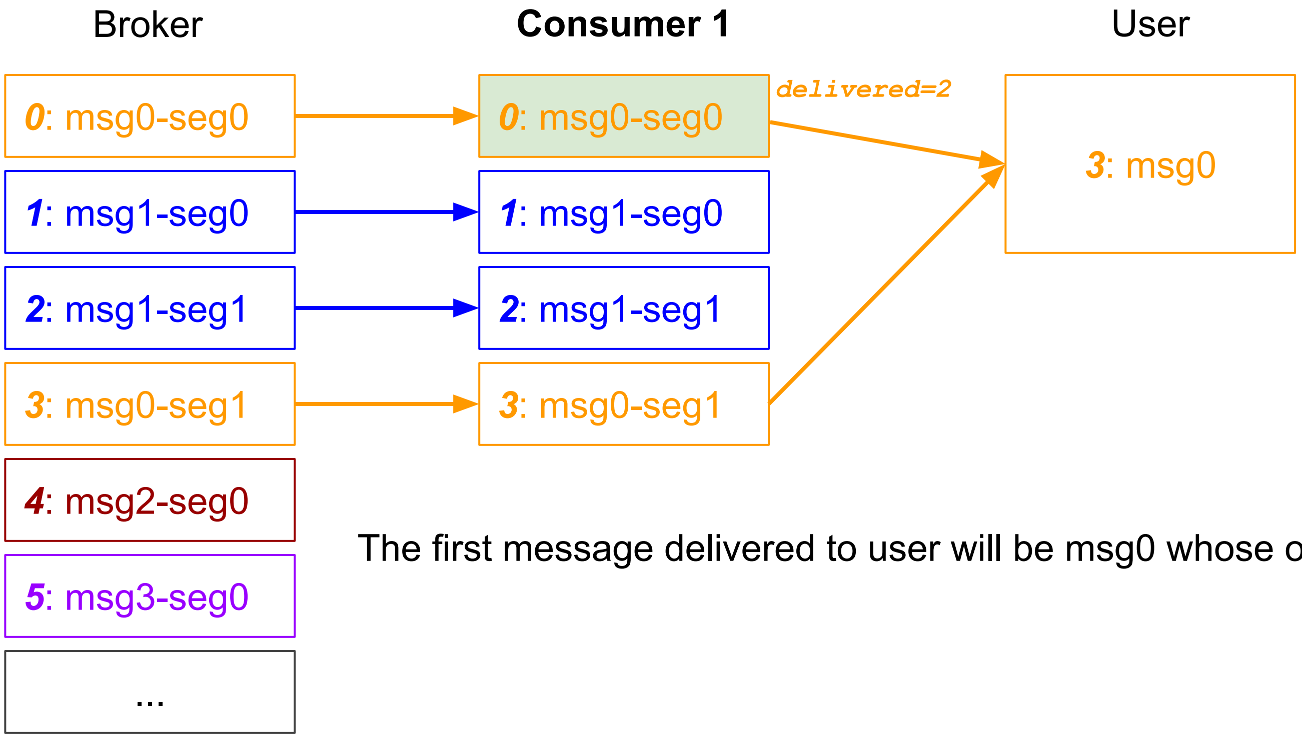
- New owner consumer 1 resumes reading from msg0-seg0
- Consumer 1 receives the committed metadata ***`{delivered=2}`***

Rebalance and duplicates handling



- msg1.offset <= delivered
- Consumer 1 will NOT deliver msg1 again to the user

Rebalance and duplicates handling



The first message delivered to user will be msg0 whose offset is **3**

A closer look at large message handling

- The offset of a large message
- Producer callback
- Offset tracking
- Rebalance and duplicates handling
- **Memory management**
- Performance overhead
- Compatibility with existing messages

Memory management

- Producer
  - No material change to memory overhead except splitting and copying the message.
- Consumer side
  - ***`buffer.capacity`***
    - The users can set maximum bytes to buffer the segments. If buffer is full, consumers evict the **oldest incomplete message**.
  - ***`expiration.offset.gap`***
    - Suppose a message has starting offset **X** and the consumer is now consuming from offset **Y**.
    - The message will be removed from the buffer if **Y - X** is greater than the ***`expiration.offset.gap`***. i.e. "timeout".

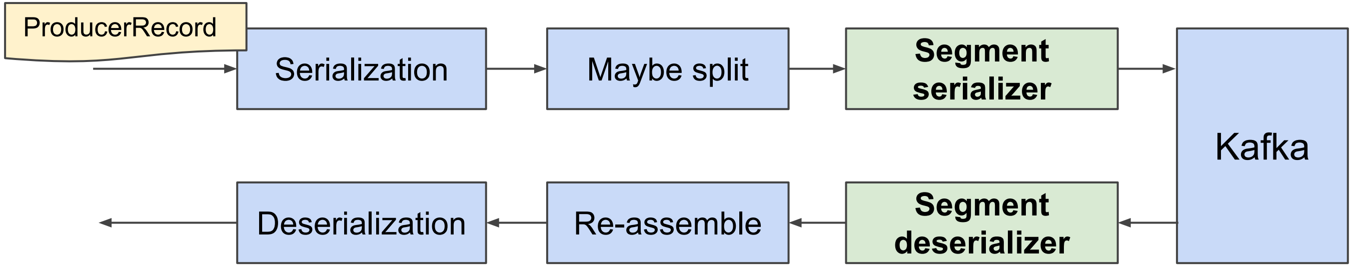
A closer look at large message handling

- The offset of a large message
- Producer callback
- Offset tracking
- Rebalance and duplicates handling
- Memory management
- **Performance overhead**
- Compatibility with existing messages

Performance Overhead

- Potentially additional segment serialization/deserialization cost
    - Default segment serde is cheap

```
{
    // segment fields
    public final UUID messageId;
    public final int sequenceNumber;
    public final int numberOfSegments;
    public final int messageSizeInBytes;
    public final ByteBuffer payload;
}
```



ProducerRecord

Serialization

Maybe split

Segment serializer

Kafka

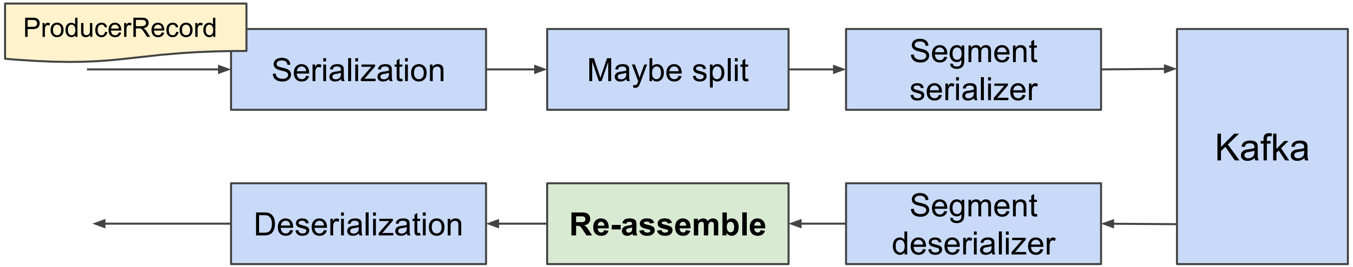
Segment deserializer

Re-assemble

Deserialization

Performance Overhead

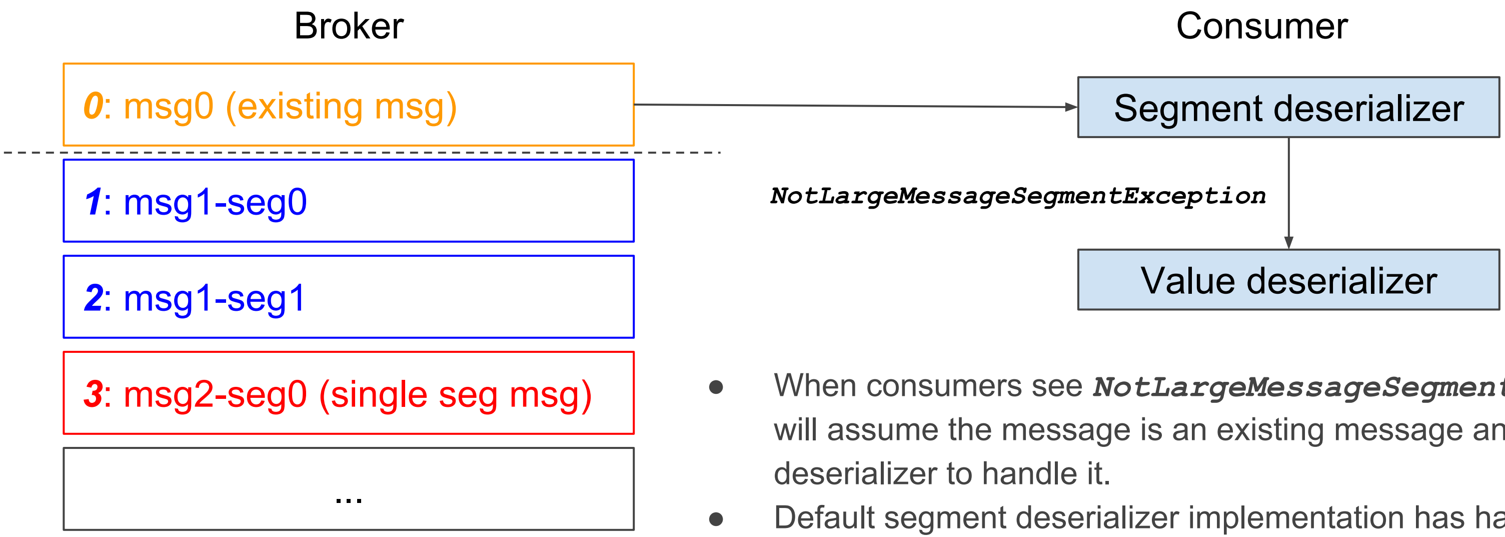
- Additional memory footprint in consumers
  - Buffer for segments of incomplete large messages
  - Additional memory needed to track the message offsets.
    - 24 bytes per message. It takes 12 MB to track the most recent 5000 messages from 100 partitions.
    - We can choose to only track large messages if users are trustworthy.



A closer look at large message handling

- The offset of a large message
- Producer callback
- Offset tracking
- Rebalance and duplicates handling
- Memory management
- Performance overhead
- **Compatibility with existing messages**

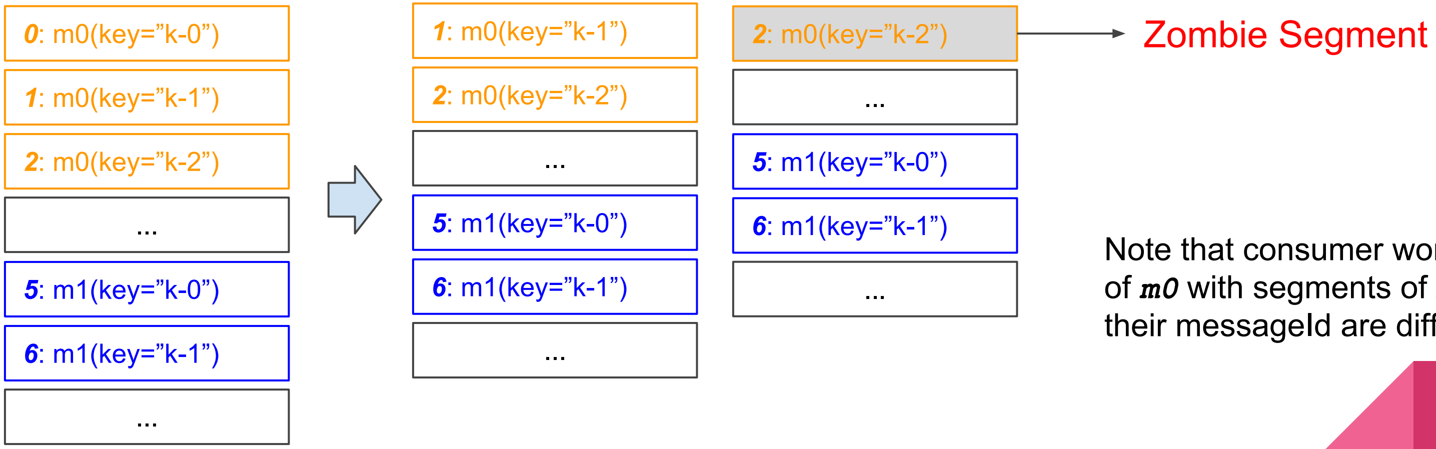
Compatibility with existing messages



- When consumers see ***NotLargeMessageSegmentException***,they will assume the message is an existing message and use value deserializer to handle it.
- Default segment deserializer implementation has handled this.
- In the segment deserializer, user implementation should throw ***NotLargeMessageSegmentException***

The answer to a question after the meetup

- Does it work for compacted topics?
  - Add suffix *"-segmentSeq"* to the key
    - It **works with a flaw** when large messages with the **same key** do **NOT interleave**



0: m0(key="k-0")

1: m0(key="k-1")

2: m0(key="k-2")

...

5: m1(key="k-0")

6: m1(key="k-1")

...

Before compaction

1: m0(key="k-1")

2: m0(key="k-2")

...

5: m1(key="k-0")

6: m1(key="k-1")

...

Scenario 1 after compaction

2: m0(key="k-2")

...

5: m1(key="k-0")

6: m1(key="k-1")

...

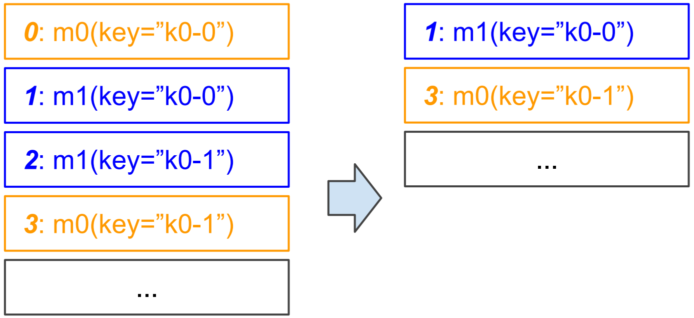
Scenario 2 after compaction

Zombie Segment

Note that consumer won't assemble segments of ***m0*** with segments of ***m1*** together because their messageId are different.

The answer to a question after the meetup

- Does it work for compacted topics?
    - Add suffix **“*-segmentSeq*”** to the key ()
        - It does **not work** when large messages with the **same key** may **interleave**



0: m0(key=”k0-0”)

1: m1(key=”k0-0”)

2: m1(key=”k0-1”)

3: m0(key=”k0-1”)

...

1: m1(key=”k0-0”)

3: m0(key=”k0-1”)

...

Note that consumer won’t assemble ***m0-seg1*** and ***m1-seg0*** together because their messageId are different

Before compaction

Failure Scenario
(Doesn’t work)

Summary

- Reference based messaging works in most cases.
- Sometimes it is handy to have in-line support for large message
  - Sporadic large messages
  - low latency
  - Small number of interleaved large messages
  - Save cost

Acknowledgements

Thanks for the great help and support from

Dong Lin
Joel Koshy
Kartik Paramasivam
Onur Karaman
Yi Pan
LinkedIn Espresso and Datastream team

Q&A